

# 1 GNU and Unix Commands (Topic 1.3)

## 1.1 Objective 1: Work Effectively on the Unix Command Line

### 1.1.1 The interactive shell and shell variables

- A *shell* provides the command prompt and interprets commands.
- A *shell variable* holds a value that is accessible to shell programs.
- `PATH` is a shell variable that contains a listing of directories that hold executable programs.
- Commands must be **bash** built-ins, found in the `PATH`, or explicitly defined in order to succeed.
- When shell variables are *exported*, they become part of the *environment*.

### 1.1.2 Entering commands

- Commands are comprised of a valid command, with or without one or more options and arguments, followed by a carriage return.
- Interactive commands can include looping structures more often used in shell scripts.

### 1.1.3 Command history, editing, and substitution

- Shell sessions can be viewed as a conversation. History, expansion, and editing make that dialogue more productive.
- Commands can be reissued, modified, and edited. Examples are shown in [Table 1](#).
- Command substitution allows the *result* of a command to be placed into a shell variable.

History Type	Examples
Expansion	!! !n ^string1^string2
Editing	<b>Ctrl-P</b> , previous line <b>Ctrl-K</b> , kill to end of line <b>Ctrl-Y</b> , paste (yank) text
Substitution	VAR=\$( command )

### 1.1.4 Recursive execution

- Many commands contain either a **-r** or **-R** option for recursive execution through a directory hierarchy.
- The **find** command is inherently recursive, and is intended to descend through directories looking for files with certain attributes or executing commands.

## 1.2 Objective 2: Process Text Streams Using Text Processing Filters

The following programs modify or manipulate text from files and standard input:

**cut** [ *files* ]

Cut out selected columns or fields from one or more *files*.

**expand** *files*

Convert tabs to spaces in *files*.

**fmt** [ *files* ]

Format text in *files* to a specified width by filling lines and removing newline characters.

**head** [ *files* ]

Print the first few lines of *files*.

**join** *file1 file2*

Print a line for each pair of input lines, one each from *file1* and *file2*, that have identical *join fields*.

**nl** [ *files* ]

Number the lines of *files*, which are concatenated in the output.

**od** [ *files* ]

Dump *files* in octal, hexadecimal, ASCII, and other formats.

**paste** *files*

Paste together corresponding lines of one or more *files* into vertical columns.

**pr** [ *file* ]

Convert a text *file* into a paginated, columnar version, with headers and page fills.

**split** [ *infile* ] [ *outfile* ]

Split *infile* into a specified number of line groups; the output will go into a succession of files of *outfileaa*, *outfileab*, and so on.

**tac** [ *file* ]

Print *file* to standard output in reverse line order.

**tail** [ *files* ]

Print the last few lines of one or more *files*.

**tr** [ *string1* [ *string2* ] ]

Translate characters by mapping from *string1* to the corresponding character in *string2*.

**wc** [ *files* ]

Print counts of characters, words, and lines for *files*.

### 1.2.1 The stream editor, sed

**sed** is a popular text-filtering program found on nearly every Unix system; it has the following syntax:

**sed** *command* [ *files* ]

**sed -e** *command1* [-e *command2*] [ *files* ]

**sed -f** *script* [ *files* ]

Execute **sed** *commands*, or those found in *script*, on standard input or *files*.

## 1.3 Objective 3: Perform Basic File Management

- Filesystem creation is called *formatting*, which prepares a disk partition for use. Linux usually uses the native *ext2* (second extended) filesystem.
- The Linux filesystem is arranged into a hierarchical structure anchored at the *root directory*, or */*. Beneath this is a tree of directories and files.
- Identification information for a filesystem object is stored in its *inode*, which holds location, modification, and security information. Filesystems are created with a finite number of inodes.

### 1.3.1 File and directory management commands

The following commands are essential for the management of files and directories:

**cp** *file1 file2*

**cp** *files directory*

Copy *file1* to *file2*, or copy *files* to *directory*.

**mkdir** *directories*

Create one or more *directories*.

**mv** *source target*

Move or rename files and directories.

**rm** *files*

Delete one or more *files* from the filesystem. When used recursively (with the **-r** option), **rm** also removes directories.

**rmdir** *directories*

Delete *directories*, which must be empty.

#### **touch files**

Change the access and/or modification times of *files* by default to the present time.

### **1.3.2 File-naming wildcards**

Wildcards (also called *file globs*) allow the specification of many files at once. A list of commonly used wildcards can be found in [Table 10-2](#).

<b>Wildcard</b>	<b>Function</b>
*	Match zero or more characters.
?	Match exactly one character.
[ characters ]	Match any single character from among <i>characters</i> listed between brackets
[ !characters ]	Match any single character other than <i>characters</i> listed between brackets.
[ a-z ]	Match any single character from among the range of characters listed between brackets.
[ !a-z ]	Match any single character from among the characters not in the range listed between brackets.
{ frag1, frag2, frag3, ... }	Brace expansion: create strings <i>frag1</i> , <i>frag2</i> , and <i>frag3</i> , etc., such that { <i>file_one</i> , <i>two</i> , <i>three</i> } yields <i>file_one</i> , <i>file_two</i> , and <i>file_three</i> .

## **1.4 Objective 4: Use Unix Streams, Pipes, and Redirects**

- A central concept for Linux and Unix systems is that *everything is a file*.
- Many system devices are represented in the filesystem using a *device file*, such as */dev/ttyS0* for a serial port.

### **1.4.1 Standard I/O**

- The shell provides the *standard I/O* capability, offering three default file descriptors to running programs.
- *Standard input (stdin)* is a text input stream, by default attached to the keyboard.
- *Standard output (stdout)* is an output stream for normal program output.
- *Standard error (stderr)* is an additional output stream meant for error messages.

### **1.4.2 Pipes and redirection**

- It is possible to tie the output of one program to the input of another. This is known as a *pipe* and is created by joining commands using the pipe symbol (`|`).
- Pipes are a special form of *redirection*, which allows you to manage the origin of input streams and the destination of output streams. Redirection syntax for various shells differs slightly. See [Table 10-3](#) for examples of common redirection operators.

<b>Table 10-3. Common Redirection Operators</b>	
<b>Redirection Function</b>	<b>Syntax for bash</b>
Send <i>stdout</i> to <i>file</i> .	<code>\$ cmd &gt; file</code> <code>\$ cmd 1&gt; file</code>
Send <i>stderr</i> to <i>file</i> .	<code>\$ cmd 2&gt; file</code>
Send both <i>stdout</i> and <i>stderr</i> to <i>file</i> .	<code>\$ cmd &gt; file 2&gt;&amp;1</code>
Receive <i>stdin</i> from <i>file</i> .	<code>\$ cmd &lt; file</code>
Append <i>stdout</i> to <i>file</i> .	<code>\$ cmd &gt;&gt; file</code> <code>\$ cmd 1&gt;&gt; file</code>
Append <i>stderr</i> to <i>file</i> .	<code>\$ cmd 2&gt;&gt; file</code>
Append both <i>stdout</i> and <i>stderr</i> to <i>file</i> .	<code>\$ cmd &gt;&gt; file 2&gt;&amp;1</code>
Pipe <i>stdout</i> from <i>cmd1</i> to <i>cmd2</i> .	<code>\$ cmd1   cmd2</code>
Pipe <i>stdout</i> and <i>stderr</i> from <i>cmd1</i> to <i>cmd2</i> .	<code>\$ cmd1 2&gt;&amp;1   cmd2</code>
Pipe <i>stdout</i> from <i>cmd1</i> to <i>cmd2</i> while simultaneously writing it to <i>file1</i> using <i>tee</i> .	<code>\$ cmd1 tee file1   cmd2</code>

## 1.5 Objective 5: Create, Monitor, and Kill Processes

- Processes have:
  - A lifetime.
  - A PID.
  - A UID.
  - A GID.
  - A parent process.
  - An environment.
  - A current working directory.

### 1.5.1 Monitoring commands

#### **ps**

Generate a one-time snapshot of the current processes on standard output.

#### **pstree**

Display a hierarchical list of processes in a tree format.

#### **top**

Generate a continuous, formatted, real-time process activity display on a terminal or in a terminal window.

### 1.5.2 Signaling processes

- Processes listen for *signals* sent by the kernel or users using the **kill** command:

#### **kill**

- *sigspec* [ *pids* ]

Send *sigspec* to *pids*.

- Common **kill** signals are listed in [Table 10-4](#).

Signal	Number	Meaning
HUP	1	Hangup, reread configuration.
INT	2	Interrupt, stop running.
KILL	9	Stop immediately.
TERM	15	Terminate nicely.
TSTP	18	Stop executing, ready to continue.

### 1.5.3 Shell job control

Shells can run processes in the *background*, where they execute on their own, or in the *foreground*, attached to a terminal. Each process handled in this way is known as a *job*.

Jobs are manipulated using job control commands:

**bg** [ *jobspec* ]

Place *jobspec* in the background as if it had been started with **&**.

**fg** [ *jobspec* ]

Place *jobspec* in the foreground, making it the current job.

**jobs** [ *jobspecs* ]

List *jobspecs* on standard output.

## 1.6 Objective 6: Modify Process Execution Priorities

- A process' *execution priority* is managed by the kernel.
- You can bias the execution priority by specifying a *nice number* in the range of -20 to +19 (default is 0).
- Positive nice numbers reduce priority; negative nice numbers increase priority and are reserved for the superuser.

**nice** – *adjustment* [ *command* ]

Apply nice number *adjustment* to the process created to run *command*.

**renice** [ + | - ] *nicenumber targets*

Alter the *nicenumber*, and thus the scheduling priority, of one or more running *target* processes.

## 1.7 Objective 7: Perform Searches of Text Files Making Use of Regular Expressions

- *Regular expressions* are used to match text. The term is used to describe the loosely defined text-matching language as well as the patterns themselves. A regular expression is often called a *regex* or a *regexp*.
- Regular expressions are made up of *metacharacters* (with special meaning) and *literals* (everything that is not a metacharacter).

### 1.7.1 Position anchors

These operators match line position:

**^**

Match the beginning of a line.

**\$**

Match the end of a line.

### 1.7.2 Character sets

These operators match text:

[ abc ]

[ a-z ]

Match any single character from among listed characters or from among the characters comprising a range.

[ ^abc ]

[ ^a-z ]

Match any single character not among listed characters or ranges.

\< word\>

Match *words* bounded by whitespace.

. (A single period, or dot)

Match any single character except a *newline*.

\

Turn off (escape) the special meaning of a metacharacter that follows.

### 1.7.3 Modifiers

These operators modify the way other operators are interpreted:

\*

Match zero or more of the character that precedes it.

?

Match zero or one instance of the preceding *regex*.

+

Match one or more instances of the preceding *regex*.

\{ *n*, *m* \}

Match a range of occurrences of the single character or *regex* that precedes this construct.

|

Match the character or expression to the left or right of the vertical bar.

## 3.2 Objective 2: Process Text Streams Using Text-Processing Filters

Many of the commands on Linux systems are intended to be used as *filters*, which modify text in helpful ways. Text fed into the command's standard input or read from files is modified in some useful way and sent to standard output or to a new file.

Multiple commands can be combined to produce *text streams*, which are modified at each step in a pipeline formation. This section describes basic use and syntax for the filtering commands important for Exam 101. Refer to a Linux command reference for full details on each command and the many other available commands.

***cut***

---

### Syntax

```
cut options [files]
```

## Description

Cut out (that is, print) selected columns or fields from one or more *files*. The source file is not changed. This is useful if you need quick access to a vertical slice of a file. By default, the slices are delimited by a **tab**.

## Frequently used options

**-b** *list*

Print bytes in *list* positions.

**-c** *list*

Print characters in *list* columns.

**-d** *delim*

Set field delimiter for **-f**.

**-f** *list*

Print *list* fields.

## Examples

Show usernames (in the first colon-delimited field) from */etc/passwd*:

```
$ cut -d: -f1 /etc/passwd
```

Show first column of */etc/passwd*:

```
$ cut -c 1 /etc/passwd
```

## ***expand***

---

## Syntax

```
expand [options] files
```

## Description

Convert tabs to spaces. Sometimes the use of tab characters can make output that is attractive on one output device look bad on another. This command eliminates tabs and

replaces them with the equivalent number of spaces. By default, tabs are assumed to be eight spaces apart.

### Frequently used options

**-t** *tabs*

Specify tab stops, in place of default 8.

**-i**

Initial; convert only at start of lines.

***fmt***

---

### Syntax

```
fmt [options] [files]
```

### Description

Format text to a specified width by filling lines and removing newline characters. Multiple *files* from the command line are concatenated.

### Frequently used options

**-u**

Use uniform spacing: one space between words and two spaces between sentences.

**-w** *width*

Set line width to *width*. The default is 75 characters.

***head***

---

### Syntax

```
head [options] [files]
```

## Description

Print the first few lines of one or more files (the "head" of the file or files). When more than one file is specified, a header is printed at the beginning of each file, and each is listed in succession.

## Frequently used options

### **-c *n***

Print the first *n* bytes, or if *n* is followed by *k* or *m*, print the first *n* kilobytes or megabytes, respectively.

### **-l *n***

Print the first *n* lines. The default is 10.

## ***join***

---

## Syntax

```
join [options] file1 file2
```

## Description

Print a line for each pair of input lines, one each from *file1* and *file2*, that have identical *join fields*. This function could be thought of as a very simple database table join, where the two files share a common index just as two tables in a database would.

## Frequently used options

### **-j1 *field***

Join on *field* of file1.

### **-j2 *field***

Join on *field* of file2.

### **-j *field***

Join on *field* of both *file1* and *file2*.

## Example

Suppose *file1* contains the following:

```
1 one
2 two
3 three
```

and *file2* contains:

```
1 11
2 22
3 33
```

Issuing the command:

```
$ join -j 1 file1 file2
```

yields the following output:

```
1 one 11
2 two 22
3 three 33
```

***nl***

---

## Syntax

```
nl [options] [files]
```

## Description

Number the lines of *files*, which are concatenated in the output. This command is used for numbering lines in the body of text, including special header and footer options normally excluded from the line numbering. The numbering is done for each *logical page*, which is defined as having a header, a body, and a footer. These are delimited by the special strings `\:\:\:`, `\:\:`, and `\:`, respectively.

## Frequently used options

### **-b** *style*

Set body numbering style to *style*, **t** by default.

### **-f** *style*

Set footer number style to *style*, **n** by default.

### **-h** *style*

Set header numbering style to *style*, **n** by default.

Styles can be in these forms:

**A**

Number all lines.

**t**

Only number non-empty lines.

**n**

Do not number lines.

**p***REGEXP*

Only number lines that contain a match for regular expression *REGEXP*.

## Example

Suppose file *file1* contains the following text:

```
\:\:\:
```

```
header
```

```
\:\:
```

```
line1
```

```
line2
```

```
line3
```

```
\:
```

```
footer
```

```
\:\:\:
```

```
header
```

```
\:\:
```

```
line1
```

```
line2
```

```
line3
```

```
\:
```

```
footer
```

If the following command is given:

```
$ nl -h a file1
```

the output would yield numbered headers and body lines but no numbering on footer lines. Each new header represents the beginning of a new logical page and thus a restart of the numbering sequence:

```
1 header
```

```
2 line1
3 line2
4 line3
```

```
footer
```

```
1 header
```

```
2 line1
3 line2
4 line3
```

```
footer
```

## ***od***

---

### **Syntax**

```
od [options] [files]
```

### **Description**

Dump files in octal and other formats. This program prints a listing of a file's contents in a variety of formats. It is often used to examine the byte codes of binary files but can be used on any file or input stream. Each line of output consists of an octal byte offset from the start of the file followed by a series of tokens indicating the contents of the file.

Depending on the options specified, these tokens can be ASCII, decimal, hexadecimal, or octal representations of the contents.

### **Frequently used options**

#### **-t *type***

Specify the *type* of output. Typical types include:

#### **A**

Named character

#### **c**

ASCII character or backslash escape

**O**

Octal (the default)

**x**

Hexadecimal

## Example

If *file1* contains:

```
a1\n
```

```
A1\n
```

where `\n` stands for the newline character. The `od` command specifying named characters yields the following output:

```
$ od -t a file1
00000000  a  1  \n  A  1  \n
00000006
```

A slight nuance is the ASCII character mode. This `od` command specifying named characters yields the following output with backslash-escaped characters rather than named characters:

```
$ od -t c file1
00000000  a  1  \n  A  1  \n
00000006
```

With numeric output formats, you can instruct `od` on how many bytes to use in interpreting each number in the data. To do this, follow the type specification by a decimal integer. This `od` command specifying single-byte hex results yields the following output:

```
$ od -t x1 file1
00000000  61 31 0a 41 31 0a
00000006
```

Doing the same thing in octal notation yields:

```
$ od -t o1 file1
00000000  141 061 012 101 061 012
00000006
```

If you examine an ASCII chart with hex and octal representations, you'll see that these results match those tables.

***paste***

---

## Syntax

```
paste [options] files
```

## Description

Paste together corresponding lines of one or more *files* into vertical columns.

## Frequently used options

### **-d'n'**

Separate columns with character *n* in place of the default **tab**.

### **-s**

Merge lines from one file into a single line. When multiple files are specified, their contents are placed on individual lines of output, one per file.

For the following three examples, *file1* contains:

```
1
2
3
```

and *file2* contains:

```
A
B
C
```

## Example 1

A simple paste creates columns from each file in standard output:

```
$ paste file1 file2
1    A
2    B
3    C
```

## Example 2

The column separator option yields columns separated by the specified character:

```
$ paste -d'@' file1 file2
1@A
2@B
3@C
```

### Example 3

The single-line option (-s) yields a line for each file:

```
$ paste -s file1 file2
1    2    3
A    B    C
```

## ***pr***

---

### Syntax

```
pr [options] [file]
```

### Description

Convert a text *file* into a paginated, columnar version, with headers and page fills. This command is convenient for yielding nice output, such as for a line printer from raw uninteresting text files. The header will consist of the date and time, the filename, and a page number.

### Frequently used options

#### **-d**

Double space.

#### **-h** *header*

Use *header* in place of the filename in the header.

#### **-l** *lines*

Set page length to *lines*. The default is 66.

#### **-o** *width*

Set the left margin to *width*.

## ***split***

---

## Syntax

```
split [option] [infile] [outfile]
```

## Description

Split *infile* into a specified number of line groups, with output going into a succession of files, *outfileaa*, *outfileab*, and so on (the default is *xaa*, *xab*, etc.). The *infile* remains unchanged. This command is handy if you have a very long text file that needs to be reduced to a succession of smaller files. This was often done to email large files in smaller chunks, because it was at one time considered bad practice to send single large email messages.

## Frequently used option

### **-n**

Split the *infile* into *n*-line segments. The default is 1000.

## Example

Suppose *file1* contains:

```
1 one
2 two
3 three
4 four
5 five
6 six
```

Then the command:

```
$ split -2 file1 splitout_
```

yields as output three new files, *splitout\_aa*, *splitout\_ab*, and *splitout\_ac*. The file *splitout\_aa* contains:

```
1 one
2 two
```

*splitout\_ab* contains:

```
3 three
```

```
4 four
```

and *splitout\_ac* contains:

```
5 five
```

```
6 six
```

## ***tac***

---

### **Syntax**

```
tac [file]
```

### **Description**

This command is named as an opposite for the **cat** command, which simply prints text files to standard output. In this case, **tac** prints the text files to standard output with lines in reverse order.

### **Example**

Suppose *file1* contains:

```
1 one
```

```
2 two
```

```
3 three
```

Then the command:

```
$ tac file1
```

yields as output:

```
3 three
```

```
2 two
```

```
1 one
```

## ***tail***

---

### **Syntax**

```
tail [options] [files]
```

## Description

Print the last few lines of one or more *files* (the "tail" of the file or files). When more than one file is specified, a header is printed at the beginning of each file, and each is listed in succession.

## Frequently used options

**-c** *n*

This option prints the last *n* bytes, or if *n* is followed by *k* or *m*, the last *n* kilobytes or megabytes, respectively.

**-f**

Follow the output dynamically as new lines are added to the bottom of a file.

**-n** *m*

Prints the last *m* lines. The default is 10.

**-f**

Continuously display a file as it is actively written by another process. This is useful for watching log files as the system runs.

**tr**

---

## Syntax

```
tr [options] [[string1 [string2]]
```

## Description

Translate characters from *string1* to the corresponding characters in *string2*. **tr** does *not* have file arguments and therefore must use standard input and output. If *string1* and *string2* specify ranges (a-z or A-Z), they should represent the same number of characters.

## Frequently used options

**-d**

Delete characters in *string1* from the output.

**-s**

Squeeze out repeated output characters in *string1*.

### Example 1

To change all lowercase characters in *file1* to uppercase, use either of these commands:

```
$ cat file1 | tr a-z A-Z
```

or:

```
$ tr a-z A-Z < file1
```

### Example 2

To suppress repeated "a" characters from *file1*:

```
$ cat file1 | tr -s a
```

### Example 3

To remove all "a," "b," and "c" characters from *file1*:

```
$ cat file1 | tr -d abc
```

## WC

---

### Syntax

```
wc [options] [files]
```

### Description

Print counts of characters, words, and lines for *files*. When multiple files are listed, statistics for each file output on a separate line with a cumulative total output last.

### Frequently used options

**-c**

Print the character count only.

**-l**

Print the line count only.

**-w**

Print the word count only.

### Example 1

Show all counts and totals for *file1*, *file2*, and *file3*:

```
$ wc file[123]
```

### Example 2

Count the number of lines in *file1*:

```
$ wc -l file1
```

## **xargs**

---

### Syntax

```
xargs [options] [command] [initial-arguments]
```

### Description

Execute *command* followed by its optional *initial-arguments* and append additional arguments found on standard input. Typically, the additional arguments are filenames in quantities too large for a single command line. **xargs** runs *command* multiple times to exhaust all arguments on standard input.

### Frequently used options

**-n maxargs**

Limit the number of additional arguments to *maxargs* for each invocation of *command*.

**-p**

Interactive mode. Prompt the user for each execution of *command*.

### Example

Use **grep** to search a long list of files, one by one, for the word "linux":

```
$ find / -type f | xargs -n 1 grep linux
```

**find** searches for normal files (**-type f**) starting at the root directory. **xargs** executes **grep** once for each of them due to the **-n 1** option.

### 3.2.1 The Stream Editor, **sed**

Another filtering program found on nearly every Unix system is **sed**, the *stream editor*. It is called a stream editor because it is intended as a filter, with text usually flowing from standard input, through the utility, to standard output. Unlike the previously listed commands, **sed** is a programmable utility with a range of capabilities. During processing, **sed** interprets instructions from a *sed script*, processing the text according to those instructions. The script may be a single command or a longer list of commands. It is important to understand **sed** and its use for Exam 101, although detailed knowledge is not required or offered in this brief introduction.

The **sed** utility is usually used either to automate repetitive editing tasks or to process text in pipes of Unix commands (see Objective 4). The scripts that **sed** executes can be single commands or more complex lists of editing instructions. It is invoked using one of the following methods.

#### **sed**

---

#### Syntax

```
sed [options] 'command1' [files]
sed [options] -e 'command1' [-e 'command2'...] [files]
sed [options] -f script [files]
```

#### Description

The first form invokes **sed** with a one-line *command1*. The second form invokes **sed** with two (or more) commands. Note that in this case the **-e** parameter is required for all commands specified. The commands are specified in quotes to prevent the shell from interpreting and expanding them. The last form instructs **sed** to take editing commands from file *script* (which does not need to be executable). In all cases, if *files* are not

specified, input is taken from standard input. If multiple *files* are specified, the edited output of each successive file is concatenated.

## Frequently used options

### **-e** *cmd*

The next argument is a command. This is not needed for single commands but is required for all commands when multiple commands are specified.

### **-f** *file*

The next argument is a script.

### **-g**

Treat all substitutions as global.

The **sed** utility operates on text through the use of *addresses* and *editing commands*. The address is used to locate lines of text to be operated upon, and editing commands modify text. During operation, each line (that is, text separated by newline characters) of input to **sed** is processed individually and without regard to adjacent lines. If multiple editing commands are to be used (through the use of a script file or multiple **-e** options), they are all applied in order to each line before moving on to the next line.

Input to **sed** can come from standard input or from *files*. When input is received from standard input, the original versions of the input text are lost. However, when input comes from files, the files themselves are *not changed* by **sed**. The output of **sed** represents a modified version of the contents of the files but does not affect them.

## Addressing

Addresses in **sed** locate lines of text to which commands will be applied. The addresses can be:

- A line number (note that **sed** counts lines continuously across multiple input files).
- A line number with an interval. The form is  $n\sim s$ , where  $n$  is the starting line number and  $s$  is the step, or interval, to apply. For example, to match every odd line in the input, the address specification would be  $1\sim 2$  (start at line 1 and match every two lines thereafter). This feature is a GNU extension to **sed**.
- The symbol **\$**, indicating the last line of the last input file.
- A regular expression delimited by forward slashes (*/regex/*). See Objective 7 for more information on using regular expressions.

Zero, one, or two such addresses can be used with a **sed** command. If no addresses are given, commands are applied to all input lines by default. If a single address is given, commands are applied only to a line or lines matching the address. If two comma-separated addresses are given, an inclusive range is implied. Finally, any address may be followed by the **!** character, and commands are applied to lines that *do not match* the address.

## Commands

The **sed** command immediately follows the address specification if present. Commands generally consist of a single letter or symbol, unless they have arguments. Following are some basic **sed** editing commands to get you started.

**d**

Delete lines.

**s**

Make substitutions. This is a very popular **sed** command. The syntax is:

```
s/pattern/replacement/[flags]
```

The following *flags* can be specified for the **s** command:

**g**

Replace all instances of *pattern*, not just the first.

**n**

Replace *n* th instance of *pattern*; the default is 1.

**p**

Print the line if a successful substitution is done. Generally used with the **-n** command-line option.

**w file**

Print the line to *file* if a successful substitution is done.

**y**

Translate characters. This command works in a fashion similar to the **tr** command, described earlier.

## Example 1

Delete lines 3 through 5 of *file1*:

```
$ sed '3,5d' file1
```

## Example 2

Delete lines of *file1* that contain a # at the beginning of the line:

```
$ sed '/^#/d' file1
```

### Example 3

Translate characters:

```
y/abc/xyz/
```

Every instance of *a* is translated to *x*, *b* to *y*, and *c* to *z*.

### Example 4

Write the @ symbol for all empty lines in *file1* (that is, lines with only a newline character but nothing more):

```
$ sed 's/^$/@/' file1
```

### Example 5

Remove all double quotation marks from all lines in *file1*:

```
$ sed 's/"/g' file1
```

### Example 6

Using **sed** commands from external file *sedcmds*, replace the third and fourth double quotation marks with ( and ) on lines 1 through 10 in *file1*. Make no changes from line 11 to the end of the file. Script file *sedcmds* contains:

```
1,10{  
s/"(/3  
s/"/)/4  
}
```

The command is executed using the **-f** option:

```
$ sed -f sedcmds file1
```

This example employs the positional flag for the **s** (substitute) command. The first of the two commands substitutes ( for the third double-quote character. The next command substitutes ) for the fourth double-quote character. Note, however, that the position count is interpreted *independently* for each subsequent command in the script. This is important because each command operates on the results of the commands preceding it. In this example, since the third double quote has been replaced with (, it is no longer counted as a double quote by the second command. Thus, the second command will

operate on the *fifth* double quote character in the original *file1*. If the input line starts out with:

```
" " " " " "
```

after the first command, which operates on the third double quote, the result is:

```
" " ( " " "
```

At this point, the numbering of the double-quote characters has changed, and the fourth double quote in the line is now the fifth character. Thus, after the second command executes, the output is:

```
" " ( " ) "
```

As you can see, creating scripts with **sed** requires that the sequential nature of the command execution be kept in mind.

If you find yourself making repetitive changes to many files on a regular basis, a **sed** script is probably warranted. Many more commands are available in **sed** than are listed here.

### 3.3 Objective 3: Perform Basic File Management

This section covers basic file and directory management, including filesystems, files and directories, standard file management commands, their recursive capabilities where applicable, and wildcard patterns.

#### 3.3.1 Filesystem Objects

Nearly every operating system that has ever been devised structures its collection of stored objects in a *hierarchy*,<sup>[10]</sup> which is a tree of objects containing other objects. This hierarchy allows a sane organization of objects and allows identically named objects to appear in multiple locations -- this is essential for multiuser systems like Linux. Information about each object in the filesystem is stored in a table (which itself is part of the filesystem), and each object is numbered uniquely within that table. Although there are a few special object types on Linux systems, the two most common are *directories* and *files*.

<sup>[10]</sup> However, it wasn't so long ago that MS-DOS was "flat" and had no hierarchy.

##### 3.3.1.1 Directories and files

A directory is an object intended to contain other objects, while a file is an object intended to contain information. At the top of all Linux filesystem hierarchies is a directory depicted simply by `/`; this is known as the *root* directory.<sup>[1]</sup> Beneath `/` are named directories and files in an organized and well-defined tree. To describe these objects, you simply refer to them by name separated by the `/` character. For example, the object `ls` is an executable program stored in a directory called `/bin` under the *root* directory; it is depicted simply as `/bin/ls`.

<sup>[1]</sup> Not to be confused with the username `root`, which is separate and distinct. There's also often a directory named `/root` for the root user. Keeping `/`, `/root` and the *root user* straight in a conversation can be a challenge.

### 3.3.1.2 Inodes

The identification information for a filesystem object is known as its *inode*. Inodes carry information about objects, such as where they are located on disk, their modification time, security settings, and so forth. Each Linux *ext2* filesystem is created with a finite number of inodes, which is a number calculated based on the number of objects contained by the filesystem. Multiple objects in the filesystem can share the same inode; this concept is called *linking*.

### 3.3.1.3 File and directory management commands

Once a hierarchy is defined, there is a constant need to manage the objects in the filesystem. Objects are constantly created, read, modified, copied, moved, and deleted, and wisely managing the filesystem is one of the most important tasks of a system administrator. In this section, we discuss the basic command-line utilities used for file and directory management. While the GUI has tools for this task, the spirit of the Linux system and the requirements of Exam 101 require your understanding of these commands.

## **cp**

---

### Syntax

```
cp [options] file1 file2
cp [options] files directory
```

## Description

In the first command form, copy *file1* to *file2*. If *file2* exists and you have appropriate privileges, it will be overwritten without warning (unless you use the **-i** option). Both *file1* and *file2* can be any valid filename, either fully qualified or in the local directory. In the second command form, copy one or more *files* to *directory*. Note that the presence of multiple files *implies* that you wish to copy files to a directory. If *directory* doesn't exist, an error message will be printed. This command form can get you in trouble if you attempt to copy a single file into a directory that doesn't exist, as the command will be interpreted as the first form and you'll end up with *file2* instead of *directory*.

## Frequently used options

**-f**

Force an overwrite of existing files in the destination.

**-i**

Prompt *interactively* before overwriting destination files. It is common practice (and advised) to alias the **cp** command to **cp -i** to prevent accidental overwrites. You may find that this is already done for you for user root on your Linux system.

**-p**

Preserve all information, including owner, group, permissions, and timestamps. Without this option, the copied file or files will have the present date and time, default permissions, owner, and group.

**-r, -R**

Recursively copy directories. You may use either upper- or lowercase for this option. If *file1* is actually a directory instead of a file and the recursive option is specified, *file2* will be a copy of the entire hierarchy under directory *file1*.

**-v**

Display the name of each file verbosely before copying.

## Example 1

Copy the messages file to the local directory (specified by .):

```
$ cp /var/log/messages .
```

## Example 2

Make an identical copy, including preservation of file attributes, of directory *src* in new directory *src2*:

```
$ cp -Rp src src2
```

Copy *file1*, *file2*, *file5*, *file6*, and *file7* from the local directory into your home directory (under *bash*):

```
$ cp file1 file2 file[567] ~
```

### ***On the Exam***

Be sure to know the difference between a file destination and a directory destination and how to force an overwrite of existing objects.

## ***mkdir***

---

### **Syntax**

```
mkdir [options] directories
```

### **Description**

Create one or more *directories*. You must have write permission in the directory where *directories* are to be created.

### **Frequently used options**

#### **-m** *mode*

Set the access *mode* for *directories*.

#### **-p**

Create intervening parent directories if they don't exist.

### **Examples**

Create a read-only directory named *personal*:

```
$ mkdir -m 444 personal
```

Create a directory tree in your home directory, as indicated with a leading tilde (~), using a single command:

```
$ mkdir -p ~/dir1/dir2/dir3
```

In this case, all three directories are created. This is faster than creating each directory individually.

### ***On the Exam***

Verify your understanding of the tilde (~) shortcut for the home directory.

## ***mv***

---

### **Syntax**

```
mv [options] source target
```

### **Description**

Move or rename files and directories. For *targets* on the same filesystem (partition), moving a file doesn't relocate the contents of the file itself. Rather, the directory entry for the target is updated with the new location. For *targets* on different filesystems, such a change can't be made, so files are copied to the target location and the original sources are deleted.



Note that **mv** is used to rename files and directories, because a rename operation requires the same directory entry update as a move.

If a *target* file or directory does not exist, *source* is renamed to *target*. If a *target* file already exists, it is overwritten with *source*. If *target* is an existing directory, *source* is moved into that directory. If *source* is one or more files and *target* is a directory, the files are moved into the directory.

### **Frequently used options**

**-f**

Force the move even if *target* exists, suppressing warning messages.

**-i**

Query interactively before moving files.

### ***On the Exam***

Remember that, from the filesystem's point of view on a single partition, renaming a file and moving it to a different location are nearly identical operations. This eliminates the need for a *rename* command.

## ***rm***

---

### **Syntax**

```
rm [options] files
```

### **Description**

Delete one or more *files* from the filesystem. To remove a file, you must have write permission in the directory that contains the file, but you do not need write permission on the file itself. The **rm** command also removes directories when the **-d**, **-r**, or **-R** option is used.

### **Frequently used options**

#### **-d**

Remove directories even if they are not empty. This option is reserved for privileged users.

#### **-f**

Force removal of write-protected files without prompting.

#### **-i**

Query interactively before removing files.

#### **-r, -R**

If the *file* is a directory, recursively remove the entire directory and all of its contents, including subdirectories.

## ***rmdir***

---

### **Syntax**

```
rmdir [option] directories
```

## Description

Delete *directories*, which must be empty.

## Frequently used option

### **-p**

Remove *directories* and any intervening parent directories that become empty as a result. This is useful for removing subdirectory trees.

### ***On the Exam***

Remember that recursive remove using **rm -R** removes directories too, even if they're not empty.

## ***touch***

---

## Syntax

```
touch [options] files
```

## Description

Change the access and/or modification times of *files*. This command is used to refresh timestamps on files. Doing so may be necessary, for example, to cause a program to be recompiled using the date-dependant **make** utility.

## Frequently used options

### **-a**

Change only the access time.

### **-m**

Change only the modification time.

### **-t *timestamp***

Instead of the current time, use *timestamp* in the form of [[CC]YY]MMDDhhmm[.ss]. For example, the *timestamp* for January 12, 2001, at 6:45 p.m. is 200101121845.

### 3.3.2 File-Naming Wildcards

When working with files on the command line, you'll often run into situations in which you need to perform operations on many files at once. For example, if you are developing a C program, you may want to **touch** all of your `.c` files in order to be sure to recompile them the next time you issue the **make** utility to build your program. There will also be times when you need to move or delete all the files in a directory or at least a selected group of files. At other times, filenames may be long or difficult to type, and you'll want to find an abbreviated alternative to typing the filenames for each command you issue.

In order to make these operations simpler, all shells<sup>[12]</sup> on Linux offer *file-naming wildcards* (Table 3-3). Rather than explicitly specifying every file or typing long filenames, specifying *wildcard characters* in place of portions of the filenames can usually do the work for you. For example, the shell expands things like `*.txt` to a list of all the files that end in `.txt`. File wildcard constructs like this are called *file globs*, and their use is awkwardly called *globbing*. Using file globs to specify multiple files is certainly a convenience, and in many cases is required to get anything useful accomplished.

<sup>[12]</sup> Wildcards are expanded by the shell, not by commands. When a command is entered with wildcards included, the shell first expands all the wildcards (and other types of expansion) and passes the full result on to the command. This process is invisible to you.

**Table 3-3. Common File-Naming Wildcards**

Wildcard	Description
*	Commonly thought to "match anything." It actually will match zero or more characters (which includes "nothing!"). For example, <code>x*</code> matches files or directories <code>x</code> , <code>xy</code> , <code>xyz</code> , <code>x.txt</code> , <code>xy.txt</code> , <code>xyz.c</code> , and so on.
?	Match exactly one character. For example, <code>x?</code> matches files or directories <code>xx</code> , <code>xy</code> , <code>xz</code> , but not <code>x</code> and not <code>xyz</code> . The specification <code>x??</code> matches <code>xyz</code> , but not <code>x</code> and <code>xy</code> .
[characters]	Match any single character from among <i>characters</i> listed between the brackets. For example, <code>x[yz]</code> matches <code>xy</code> and <code>xz</code> .
[!characters]	Match any single character other than <i>characters</i> listed between the brackets. For example, <code>x[!yz]</code> matches <code>xa</code> and <code>xI</code> but does not match <code>xy</code> and does not match <code>xz</code> .
[a-z]	Match any single character from among the range of characters listed between the brackets and indicated by the dash (the dash character is not matched). For example, <code>x[0-9]</code> matches <code>x0</code> and <code>xI</code> , but does not match <code>xx</code> . Note that to match both upper- and lowercase letters, <sup>[13]</sup> you specify <code>[a-zA-Z]</code> . Using <code>x[a-zA-Z]</code> matches <code>xa</code> and <code>xA</code> .
[!a-z]	Match any single character from among the characters not in the range listed between the brackets.
{ frag1, frag2, frag3... }	Create strings <code>frag1</code> , <code>frag2</code> , <code>frag3</code> , etc. For example, <code>file_{one,two,three}</code> yields the strings <code>file_one</code> , <code>file_two</code> , and <code>file_three</code> . This is a special operator named <i>brace</i>

*expansion* that can be used to match filenames but isn't specifically a file wildcard operator and does not examine directories for existing files to match. Instead, it will expand *any string*.

For example, it can be used with **echo** to yield strings totally unrelated to existing filenames:

```
$ echo string_{a,b,c}
string_a string_b string_c
```

<sup>[13]</sup> Linux filenames are case-sensitive.

Here are a few common applications for wildcards:

- If you remember part of a filename but not the whole thing, use wildcards with the portion you remember to help find the file. For example, if you're working in a directory with a large number of files and you know you're looking for a file named for Linux, you may enter a command like this:

```
$ ls -l *linux*
```

- When working with groups of related files, wildcards can be used to help separate the groups. For example, suppose you have a directory full of scripts you've written. Some are Perl scripts, for which you've used an extension of *.pl*, and some are Python, with a *.py* extension. You may wish to separate them into new separate directories for the two languages like this:

- \$ **mkdir perl python**

- \$ **mv \*.pl perl**

- \$ **mv \*.py python**

- Wildcards match directory names as well. Suppose you have a tree of directories starting with *contracting*, where you've created a directory for each month (that is, *contracting/january*, *contracting/february*, through *contracting/december*). In each of these directories are stored invoices, named simply *invoice\_custa\_01.txt*, *invoice\_custa\_02.txt*, *invoice\_custb\_01.txt*, and so on, where *custa* and *custb* are customer names of some form. To display all of the invoices, wildcards can be used:

```
$ ls con*/*/inv*.txt
```

The first \* matches *tracting*. The second matches all directories under the *contracting* directory (*january* through *december*). The last matches all the customers and each invoice number for each customer.

See the **bash** man or info pages for additional information on how **bash** handles expansions and on other expansion forms.

### 3.4 Objective 4: Use Unix Streams, Pipes, and Redirects

Among the many beauties of the Linux and Unix systems is the notion that *everything is a file*. Things such as disk drives and their partitions, tape drives, terminals, serial ports, the mouse, and even audio are mapped into the filesystem. This mapping allows programs to interact with many different devices and files in the same way, simplifying their interfaces. Each device that uses the file metaphor is given a *device file*, which is a special object in the filesystem that provides an interface to the device. The kernel associates device drivers with various device files, which is how the system manages the illusion that devices can be accessed as if they were files. Using a terminal as an example, a program reading from the terminal's device file will receive characters typed at the keyboard. Writing to the terminal causes characters to appear on the screen. While it may seem odd to think of your terminal as a file, the concept provides a unifying simplicity to Linux and Linux programming.

#### 3.4.1 Standard I/O and Default File Descriptors

*Standard I/O* is a capability of the shell, used with all text-based Linux utilities to control and direct program input, output, and error information. When a program is launched, it is automatically provided with three *file descriptors*. File descriptors are regularly used in programming and serve as a "handle" of sorts to another file. Standard I/O creates the following file descriptors:

##### Standard input (abbreviated *stdin*)

This file descriptor is a text input stream. By default it is attached to your keyboard. When you type characters into an interactive text program, you are feeding them to standard input. As you've seen, some programs take one or more

filenames as command-line arguments and ignore standard input. Standard input is also known as *file descriptor 0*.

#### Standard output (abbreviated *stdout*)

This file descriptor is a text output stream for normal program output. By default it is attached to your terminal (or terminal window). Output generated by commands is written to standard output for display. Standard output is also known as *file descriptor 1*.

#### Standard error (abbreviated *stderr*)

This file descriptor is also a text output stream, but it is used exclusively for errors or other information unrelated to the successful results of your command. By default standard error is attached to your terminal just like standard output. This means that standard output and standard error are commingled in your display, which can be confusing. You'll see ways to handle this later. Standard error is also known as *file descriptor 2*.

Standard output and standard error are separated because it is often useful to process normal program output differently than errors.

The standard I/O file descriptors are used in the same way as those created during program execution to read and write disk files. They enable you to tie commands together with files and devices, managing command input and output in exactly the way you desire. The difference is they are provided to the program by the shell by default and do not need to be explicitly created.

### 3.4.2 Pipes

From a program's point of view there is no difference between reading text data from a file and reading it from your keyboard. Similarly, writing text to a file and writing text to a display are equivalent operations. As an extension of this idea, it is also possible to tie the output of one program to the input of another. This is accomplished using a *pipe* (`|`) to join two or more commands together. For example:

```
$ grep "01523" order* | less
```

This command searches through all files whose names begin with **order** to find lines containing the word *01523*. By creating this pipe, the standard output of **grep** is sent to the standard input of **less**. The mechanics of this operation are handled by the shell and are invisible to the user. Pipes can be used in a series of many commands. When more than two commands are put together, the resulting operation is known as a *pipeline* or *text stream*, implying the flow of text from one command to the next.

As you get used to the idea, you'll find yourself building pipelines naturally to extract specific information from text data sources. For example, suppose you wish to view a sorted list of inode numbers from among the files in your current directory. There are many ways you could achieve this. One way would be to use **awk** in a pipeline to extract the inode number from the output of **ls**, then send it on to the **sort** command and finally to a pager for viewing:<sup>[14]</sup>

<sup>[14]</sup> Don't worry about the syntax or function of these commands at this point.

```
$ ls -i * | awk '{print $1}' | sort -nu | less
```

The pipeline concept in particular is a feature of Linux and Unix that draws on the fact that your system contains a diverse set of tools for operating on text. Combining their capabilities can yield quick and easy ways to extract otherwise hard to handle information.

### 3.4.3 Redirection

Each pipe symbol in the previous pipelines example instructs the shell to feed output from one command into the input of another. This action is a special form of *redirection*, which allows you to manage the origin of input streams and the destination of output streams. In the previous example, individual programs are unaware that their output is being handed off to or from another program because the shell takes care of the redirection on their behalf.

Redirection can also occur to and from files. For example, rather than sending the output of an inode list to the pager **less**, it could easily be sent directly to a file with the **>** redirection operator:

```
$ ls -i * | awk '{print $1}' | sort -nu > in.txt
```

By changing the last redirection operator, the shell creates an empty file (*in.txt*), opens it for writing, and the standard output of **sort** places the results in the file instead of on the screen. Note that, in this example, anything sent to standard error is still displayed on the screen.

Since the **>** redirection operator *creates* files, the **>>** redirection operator can be used to append to existing files. For example, you could use the following command to append a one-line footnote to *in.txt*:

```
$ echo "end of list" >> in.txt
```

Since *in.txt* already exists, the quote will be appended to the bottom of the existing file. If the file didn't exist, the >> operator would create the file and insert the text "end of list" as its contents.

It is important to note that when creating files, the output redirection operators are interpreted by the shell *before* the commands are executed. This means that any output files created through redirection are opened first. For this reason, you cannot modify a file in place, like this:

```
$ grep "stuff" file1 > file1 # don't do it!
```

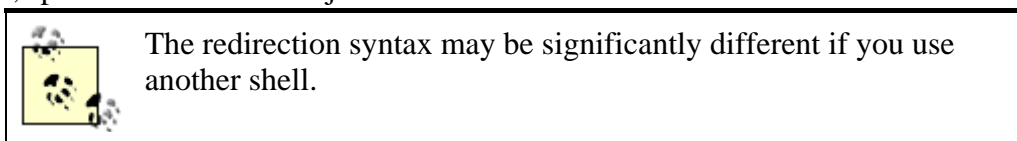
If *file1* contains something of importance, this command would be a disaster because an empty *file1* would overwrite the original. The **grep** command would be last to execute, resulting in a complete data loss from the original *file1* file because the file that replaced it was empty. To avoid this problem, simply use an intermediate file and *then* rename it:

```
$ grep "stuff" file1 > file2
$ mv file2 file1
```

Standard input can also be redirected. The input redirection operator is <. Using a source other than the keyboard for a program's input may seem odd at first, but since text programs don't care about where their standard input streams originate, you can easily redirect input. For example, the following command will send a mail message with the contents of the file *in.txt* to user *jdean*:

```
$ Mail -s "inode list" jdean < in.txt
```

Normally, the **Mail** program prompts the user for input at the terminal. However with standard input redirected *from* the file *in.txt*, no user input is needed and the command executes silently. [Table 3-4](#) lists the common standard I/O redirections for the **bash** shell, specified in the LPI Objectives.



<b>Table 3-4. Standard I/O Redirections for the bash shell</b>	
Redirection Function	Syntax for bash
Send <i>stdout</i> to <i>file</i> .	\$ <b>cmd</b> > <b>file</b> \$ <b>cmd</b> 1> <b>file</b>
Send <i>stderr</i> to <i>file</i> .	\$ <b>cmd</b> 2> <b>file</b>
Send both <i>stdout</i> and <i>stderr</i> to <i>file</i> .	\$ <b>cmd</b> > <b>file</b> 2>&1
Send <i>stdout</i> to <i>file1</i> and <i>stderr</i> to <i>file2</i> .	\$ <b>cmd</b> > <b>file1</b> 2> <b>file2</b>
Receive <i>stdin</i> from <i>file</i> .	\$ <b>cmd</b> < <b>file</b>
Append <i>stdout</i> to <i>file</i> .	\$ <b>cmd</b> >> <b>file</b> \$ <b>cmd</b> 1>> <b>file</b>
Append <i>stderr</i> to <i>file</i> .	\$ <b>cmd</b> 2>> <b>file</b>

Append both <i>stdout</i> and <i>stderr</i> to <i>file</i> .	\$ <code>cmd &gt;&gt; file 2&gt;&amp;1</code>
Pipe <i>stdout</i> from <i>cmd1</i> to <i>cmd2</i> .	\$ <code>cmd1   cmd2</code>
Pipe <i>stdout</i> and <i>stderr</i> from <i>cmd1</i> to <i>cmd2</i> .	\$ <code>cmd1 2&gt;&amp;1   cmd2</code>

### On the Exam

Be prepared to demonstrate the difference between filenames and command names in commands using redirection operators. Also, check the syntax on commands in redirection questions to be sure about which command or file is a data source and which is a destination.

### 3.4.4 Using the tee Command

Sometimes, you'll want to run a program and send its output to a file while at the same time viewing the output on the screen. The `tee` utility is helpful in this situation.

#### tee

---

#### Syntax

```
tee [options] files
```

#### Description

Read from standard input and write both to one or more *files* and to standard output (analogous to a tee junction in a pipe).

#### Option

**-a**

Append to *files* rather than overwriting them.

#### Example

Suppose you're running a pipeline of commands `cmd1`, `cmd2`, and `cmd3`:

```
$ cmd1 | cmd2 | cmd3 > file1
```

This sequence puts the ultimate output of the pipeline into *file1*. However, you may also be interested in the intermediate result of `cmd1`. To create a new *file\_cmd1* containing those results, use `tee`:

```
$ cmd1 | tee file_cmd1 | cmd2 | cmd3 > file1
```

The results in *file1* will be the same as in the original example, and the intermediate results of **cmd1** will be placed in *file\_cmd1*.

## 3.5 Objective 5: Create, Monitor, and Kill Processes

This Objective looks at the management of *processes*. Just as file management is a fundamental system administrator's function, the management and control of processes is also essential for smooth system operation. In most cases, processes will live, execute, and die without intervention from the user because they are automatically managed by the kernel. However, there are times that a process will die for some unknown reason and need to be restarted. Or, some process may "run wild" and consume system resources, requiring that it be terminated. You will also need to instruct running processes to perform operations, such as rereading a configuration file.

### 3.5.1 Processes

Each program running on your system is considered to be a process by the kernel. Your shell is a process, and each command you type into the shell starts one or more processes during its execution. Attributes and concepts associated with processes include:

#### *Lifetime*

Each process "lives" as it executes. Short commands such as **ls** will execute for a very short time, generate results, and terminate on their own. User programs such as web browsers run for extended periods until terminated by the user. Daemons such as web servers run continuously from boot to shutdown or restart. When a process terminates, it is said to *die* (which is why the program used to manually signal a process to stop execution is called **kill**; succinct, though admittedly morbid).

#### *Process ID (PID)*

Every process has a number assigned to it when it starts. PIDs are integer numbers unique among all running processes.

#### *User ID (UID) and Group ID (GID)*

Processes must have associated privileges, and a process' UID and GID are associated with the user who started the process. This limits the process' access to objects in the filesystem.

### Parent process

The first process started by the kernel at system start time is a program called **init**. This process has PID 1 and is the ultimate parent of all other processes on the system. Your shell is a descendant of **init** and the parent process to commands started by the shell, which are its *child* processes, or *subprocesses*.<sup>[15]</sup>

<sup>[15]</sup> Note that shell's built-in commands, such as **alias**, **bg**, **cd**, **echo**, **jobs**, and **test**, do not start a child process but are executed in the shell process itself. See the **bash** manpage for a full list of built-in commands.

### Parent process ID (parent PID)

*This is the PID of the process that created the process in question.* If that parent process has vanished, the parent PID will be 1, which is the PID of **init**.

### Environment

Each process holds a list of variables and their associated values. Collectively, this list is known as the process' *environment* and the variables are *environment variables*. The environment is inherited from the parent process unless it is replaced through the execution of startup scripts.

### Current working directory

A default directory is associated with each process, which is where the process will seek and write files unless they are explicitly specified to be elsewhere in the filesystem.



Processes are often referred to as *tasks*. Linux is a multitasking operating system, in that it runs many processes simultaneously. However, even though the terms *process* and *task* may be synonymous in this context, don't confuse multiprocessing with multitasking. Multiprocessing generally refers to systems with multiple central processing units, a definition that has little to do with system processes. Given the correct hardware and a multiprocessing kernel, Linux has multiprocessing capability.

## 3.5.2 Process Monitoring

At any time, there could be tens or even hundreds of processes running together on your Linux system. Monitoring these processes is done using three convenient utilities: **ps**, **pstree**, and **top**.

## **ps**

---

### **Syntax**

```
ps [options]
```

### **Description**

This command generates a one-time snapshot of the current processes on standard output.

### **Frequently used options**

#### **-a**

Show processes that are owned by other users and attached to a terminal. Normally, only the current user's processes are shown.

#### **-f**

"Forest" mode, which is used to display process family trees. For example, using this option will display all running child web servers (**httpd**) in a hierarchical diagram under the parent web server.<sup>[16]</sup>

<sup>[16]</sup> There is also a separate command called **pstree** that does this nicely.

#### **-l**

Long format, which includes priority, parent PID, and other information.

#### **-u**

User format, which includes usernames and the start time of processes.

#### **-w**

Wide output format, used to eliminate the default output line truncation. Useful for the **-f** option.

#### **-x**

Include processes without controlling terminals. Often needed to see daemon processes and others not started from a terminal session.

**-C** *cmd*

Display instances of command name *cmd*.

**-U** *usr*

Display processes owned by username *usr*.

## Examples

Simply entering the **ps** command with no options will yield a brief list of processes owned by you and attached to your terminal:

```
$ ps
```

Use the **-a**, **-u**, and **-x** options to include processes owned by others and not attached to terminals as well as to display them in the "user" mode. The command is valid with or without the dash:

```
$ ps -aux
```

```
$ ps aux
```

In this case, the dash is optional. However, certain **ps** options require the dash. (See the manpage for details.)

If you are interested in finding process information on a particular command, use the **-C** option. This command displays all web server processes:

```
$ ps u -C httpd
```

You'll note that the **-C** option *requires* the dash, but the **u** option won't work with it if a dash is included. This confusion exists because the **ps** command as implemented on Linux understands options in three differing forms:

### *Unix98 options*

These may be grouped and must be preceded by a dash.

### *BSD options*

These may be grouped and must *not* be used with a dash.

### *GNU long options*

These options are preceded by two dashes.

All of these option types may be freely intermixed. Instead of the **-C** option, you may wish to use **ps** with other options that you usually use and pipe the output to **grep**, searching for process names, PIDs, or anything else you know about the process:

```
$ ps -aux | grep httpd
```

In this case, the result would be the same list of **httpd** servers, as well as the **grep** command itself.

## ***pstree***

---

### **Syntax**

```
pstree [options] [pid|user]
```

### **Description**

The **pstree** command is similar to the "forest" mode of **ps -f**. This command displays a hierarchical list of processes in a tree format. **pstree** is very handy for understanding how parent/child process relationships are set up.

If *pid* is specified, the displayed tree is rooted at that process. Otherwise, it is rooted at the *init* process, which has PID 1. If *user* (a valid username) is specified, trees for all processes owned by *user* are shown. The tree is represented using characters that appear as lines, such as | for vertical lines and + for intersections (VT100 line-drawing characters, displayed as solid lines by most terminals, are optional). The output looks similar to this:

```
httpd+-httpd
      |-httpd
      |-httpd
      |-httpd
      `-httpd
```

By default, visually identical branches of the tree are merged to reduce output. Merged lines are preceded by a count indicating the actual number of similar processes. The preceding example is normally displayed on a single line:

```
httpd---5*[httpd]
```

This behavior can be turned off with the **-c** option.

### **Frequently used options**

**-a**

Display command-line arguments used to launch processes.

**-c**

Disable the compaction of identical subtrees.

**-G**

Use the VT100 line-drawing characters instead of plain characters to display the tree. This yields a much more pleasing display but may not be appropriate for printing or paging programs.

**-h**

Highlight the ancestry of the current process (usually the shell). The terminal must support highlighting for this option to be meaningful.

**-n**

The default sort order for processes with the same parent is alphanumerically by name. This option changes this behavior to a numeric sort by PID.

**-p**

Include PIDs in the output.

## Example

Display a process tree including PIDs:

```
# pstree -p
init(1)-+-atd(356)
    |-crond(370)
    |-gpm(526)
    |-httpd(540)-+-httpd(544)
    |               |-httpd(545)
    |               |-httpd(546)
    |               |-httpd(547)
    |               |-httpd(548)
    |-inetd(384)
    |-login(691)-bash(699)-startx(711)-xinit(718)
    |               +-X(719)
    |-lpd(412)
    |-mdrecoveryd(5)
    |-mingetty(692)
    |-mingetty(693)
    |-named(398)
    |-nfsd(467)---lockd(475)---rpciod(476)
    |-nfsd(468)
    |-portmap(284)
```

**top**

---

## Syntax

`top [command-line options]`

## Description

The **top** command also offers output similar to **ps**, but in a continuously updated display. This is useful in situations in which you need to watch the status of one or more processes or to see how they are using your system.

In addition, a header of useful uptime, load, CPU status, and memory information is displayed. By default, the process status output is generated with the most CPU-intensive processes at the top of the listing (and is named for the "top" processes). In order to format the screen, **top** must understand how to control the terminal display. The type of terminal (or terminal window) in use is stored in the environment variable `TERM`. If this variable is not set or contains an unknown terminal type, **top** may not execute.

## Popular command-line options

Dashes are not required for **top** options:

### **-b**

Run in batch mode. This is useful for sending output from **top** to other programs or to a file. It executes the number of iterations specified with the **-n** option and terminate. This option is also useful if **top** cannot display on the terminal type you are using.

### **-d** *delay*

Specify the *delay* in seconds between screen updates. The default is five seconds.

### **-i**

Ignore idle processes, listing only the "interesting" ones taking system resources.

### **-n** *num*

Display *num* iterations and then exit, instead of running indefinitely.

### **-q**

Run with no delay. If the user is the superuser, run with highest possible priority. This option causes **top** to update continuously and will probably consume any

idle time your CPU had. Running **top -q** as superuser will seriously affect system performance and is not recommended.

**-s**

Run in secure mode. Some of **top**'s interactive commands can be dangerous if running as the superuser. This option disables them.

## Frequently used interactive options

Once **top** is running interactively, it can be given a number of commands via the keyboard to change its behavior. These commands are single-key commands, some of which cause **top** to prompt for input:

**Ctrl-L**

Repaint the screen.

**h**

Generate a help screen.

**k**

Kill a process. You will be prompted for the PID of the process and the signal to send it (the default signal is 15, *SIGTERM*). See [Section 3.5.4](#).

**n**

Change the number of processes to show. You will be prompted to enter an integer number. The default is 0, which indicates that the screen should be filled.

**q**

Quit the program.

**r**

**Renice** a process (change its priority). You will be prompted for the PID of the process and the value to nice it to (see **nice** and **renice** in Objective 6). Entering a positive value causes a process to lose priority. If the superuser is running **top**, a negative value may be entered, causing a process to get a higher than normal priority. This command is not available in secure mode.

**s**

Change the delay in seconds between updates. You will be prompted for the delay value, which may include fractions of seconds (i.e., 0.5).

## Example 1

Simply executing **top** without options gives a full status display updated every five seconds:

```
$ top
```

Use the **q** command to quit.

### Example 2

To run **top** with a faster refresh rate, use the interval option, specified here with a one-second refresh:

```
$ top -d 1
```

### Example 3

To have **top** update constantly, you could specify **-d 0**, or use the **-q** option. Here, this feature is used to watch only nonidle processes, which will include **top** itself:

```
$ top -qi
```

### Example 4

You may wish to use **top** to log its output to a file. Use the **-b** (batch) option for this purpose. In this batch example, the **-i** option eliminates idle processes, the **-n** option, with its argument, indicates five iterations, and the **-d** option indicates a one-second interval. Results will be redirected to *file1*. This command will take five seconds to execute and does not use the optional dashes:

```
$ top bin 5 d 1 > file1
```

The single-key interactive commands can be used when **top** is running interactively. For example, if you type the **h** command, **top** yields a help screen. By entering the **n** command, **top** prompts you for the number of lines you wish to display.

#### ***On the Exam***

The parent/child relationship of the processes on a Linux system is important. Be sure to understand how these relationships work and how to view them. Note that the *init* process always has PID 1 and is the ultimate ancestor of all system processes.

Using **top** to change the "nice" (priority modifier) value for a process is discussed in Objective 6.

### 3.5.3 Signaling Active Processes

Each process running on your system listens for *signals*, simple messages sent to the process either by the kernel or by a user. The messages are sent through interprocess

communication. They are single-valued, in that they don't contain strings or command-like constructs. Instead, signals are numeric integer messages, predefined and known by processes. Most have an implied action for the process to take. When a process receives a signal, it can (or may be forced) to take action. For example, if you are executing a program from the command line that appears to hang, you may elect to type **Ctrl-C** to abort the program. This action actually sends an INTERRUPT signal to the process, telling it to stop running.

There are about 30 signals defined in Linux. Each signal has a name and a number (the number is sent to the process, the name is only for our convenience). Many signals are used by the kernel, and some are useful for users. [Table 3-5](#) lists popular signals for interactive use.

<b>Table 3-5. Frequently Used Interactive Signals</b>		
<b>Signal Name</b> <sup>[17]</sup>	<b>Number</b>	<b>Meaning and Use</b>
HUP	1	Hang up. This signal is sent automatically when you log out or disconnect a modem. It is also used by many daemons to cause the configuration file to be reread.
INT	2	Interrupt; stop running. This signal is sent when you type <b>Ctrl-C</b> .
KILL	9	Kill; stop unconditionally and immediately. Sending this signal is a drastic measure, as it cannot be ignored by the process. This is the "emergency kill" signal.
TERM	15	Terminate, nicely if possible. This signal is used to ask a process to exit gracefully.
TSTP	18	Stop executing, ready to continue. This signal is sent when you type <b>Ctrl-Z</b> . (See <a href="#">Section 3.5.5</a> for more information.)

<sup>[17]</sup> Signal names will often be specified with a "SIG" prefix. That is, signal **HUP** is the same as signal **SIGHUP**.

As you can see from [Table 3-5](#) some signals are invoked by pressing well-known key combinations such as **Ctrl-C** and **Ctrl-Z**. You can also use the **kill** command to send any message. The **kill** command is implemented both as a shell built-in command and as a standalone binary command.

## **kill**

---

### **Syntax**

```
kill [-s sigspec | -sigspec] [pids]
kill -l [signum]
```

### **Description**

In the first form, **kill** is used with an optional *sigspec*. This is a signal value, specified as either an integer or the signal name (such as **SIGHUP**, or simply **HUP**). The *sigspec* is case-insensitive but usually specified with uppercase letters. You may use **-s sigspec** or simply **-sigspec** to make up the signal value or name. If a *sigspec* is not given, then **SIGTERM** (signal 15, "exit gracefully") is assumed. The *sigspec* is followed by one or more *pids* to which the signal is to be sent. In the second form with the **-l** option, **kill** lists the valid signal names. If *signum* (an integer) is present, only the signal name for that number will be displayed.

## Examples

This command displays the signal name **SIGTERM**, the name of signal 15, and the default when **kill** is used to signal processes:

```
$ kill -l 15
```

All of these commands will send a **SIGTERM** signal to the processes with PIDs 1000 and 1001:

```
$ kill 1000 1001
```

```
$ kill -15 1000 1001
```

```
$ kill -SIGTERM 1000 1001
```

```
$ kill -sigterm 1000 1001
```

```
$ kill -TERM 1000 1001
```

```
$ kill -s 15 1000 1001
```

```
$ kill -s SIGTERM 1000 1001
```

If those two processes are playing nicely on your system, they'll comply with the **SIGTERM** signal and terminate when they're ready (after they clean up whatever they're doing). Not all processes will comply, however. A process may be hung in such a way that it cannot respond, or it may have *signal handling* code written to trap the signal you're trying to send. To force a process to die, use the strongest **kill**:

```
$ kill -9 1000 1001
```

```
$ kill -KILL 1000 1001
```

These equivalent commands send the **KILL** signal to the process, which the process cannot ignore. The process will terminate immediately without regard to closing files or other cleanup procedures. Because of this, using the **KILL** signal is a last resort.<sup>[18]</sup> See [Section 3.5.4](#).

<sup>[18]</sup> There are situations in which the **KILL** signal won't stop a process. Most of them are hardware-related, such as a process trying to write to an unavailable NFS server or waiting for a tape device to complete rewinding.

The **inetd** superdaemon will respond to the **HUP** signal by rereading its configuration file. If you've made changes to that file and want **inetd** to reconfigure itself, send it the **HUP** signal:

```
$ kill -HUP `cat /var/run/inetd.pid`
```

### ***On the Exam***

Note that **kill** is used for sending all kinds of signals, not just termination signals. Also, be aware of the difference between the PID you intend to kill and the signal you wish to send it. Since they're both integers, they can sometimes be confused.

The backward quotes are replaced by the shell with the contents of the file *inetd.pid*, which **inetd** creates when it starts.

### **3.5.4 Terminating Processes**

Based on the type of service that has failed, you can use **ps** or **top** to identify one or more processes that may have a problem. Once you know the PID for the process that's causing the problem, you can use the **kill** command to stop the process nicely with **SIGTERM** (**kill -15 [PID]**), escalating the signal to higher strengths if necessary until the process terminates.



Occasionally you may see a process displayed by **ps** or **top** that is listed as a *zombie*. These are processes that are stuck while trying to terminate and are appropriately said to be in the *zombie state*. Just as in the cult classic film *Night of the Living Dead*, you can't kill zombies, because they're already dead!

If you have a recurring problem with zombies, there may be a bug in your system software or in an application

Killing a process may also kill all of its child processes. For example, killing a shell may kill all the processes initiated from that shell, including other shells.

### **3.5.5 Shell Job Control**

Linux and most modern Unix systems offer *job control*, which is the ability of your shell (with support of the kernel) to place executing commands in the *background* where they can be executed. A program is said to be in the *foreground* when it is attached to your terminal. When executing in the background, you have no input to the process other than sending it signals. When a process is started in the background, you create a *job*. Each job is assigned a job number, starting at 1 and numbering sequentially.

The basic reason to create a background process is to keep your terminal or terminal window session free. There are many instances when a long-running program will never produce a result from standard output or standard error, and your shell will simply sit idle waiting for the program to finish. Noninteractive programs can be placed in the background by adding the **&** character to the command. For example, if you start **netscape** from the command line, you don't want the shell to sit and wait for it to terminate. The shell will respond by starting the browser in the background and will give you a new command prompt. It will also issue the job number, denoted in square brackets, along with the PID. For example:

```
$ netscape &
```

```
[1] 1748
```

Here, Netscape is started as a background process. Netscape is assigned to job 1 (as denoted by [1]), and is assigned PID 1748. If you start a program and forget the **&** character, you can still put it in the background by first stopping it by typing **Ctrl-Z**:

```
^Z
```

```
[1]+ Stopped      netscape
```

Then issue the **bg** command to restart the job in the background:

```
$ bg
```

```
[1]+ netscape &
```

Putting interactive programs in the background can be quite useful. Suppose you're logged into a remote Linux system, running Emacs in text mode. Realizing that you need to drop back to the command line, you elect not to terminate the editor but instead simply press **Ctrl-Z**. This stops Emacs and puts it in the background and returns you a command prompt.<sup>[19]</sup> When you're finished, you can go back into Emacs by issuing the **fg** command, which puts your stopped job back into the foreground.

<sup>[19]</sup> This example ignores the fact that Emacs is capable of hosting a shell itself, which would probably eliminate your need to use job control to get to the command line.

Background jobs and their status can be listed by issuing the **jobs** command. Stopped jobs can be brought to the foreground with the **fg** command and optionally placed into the background with the **Ctrl-Z** and **bg** sequence.

**bg**

---

## Syntax

```
bg [jobspec]
```

## Description

Place *jobspec* in the background, as if it had been started with **&**. If *jobspec* is not present, then the shell's notion of the *current job* is used, as indicated by the plus sign (+) in output from the *jobs* command. Using this command on a job that is stopped will allow it to run in the background.

***fg***

---

## Syntax

```
fg [jobspec]
```

## Description

This command places the specified job in the foreground, making it the current job. If *jobspec* is not present, then the shell's notion of the current job is used.

***jobs***

---

## Syntax

```
jobs [options] [jobspecs]
```

## Description

List the active jobs. If *jobspecs* are included, output is restricted to information about those jobs.

## Frequently used option

-l

Also list PIDs.

### ***On the Exam***

Be sure to know how to display background jobs and how to switch among them.

## **3.6 Objective 6: Modify Process Execution Priorities**

Certain tasks on the system require more execution time than others, and thus deserve to be allocated more frequent execution and longer time slices by the kernel. For this reason, each process is given an *execution priority*. Usually, you don't need to worry about execution priority because the kernel handles it automatically. Each process' priority level is constantly and dynamically raised and lowered by the kernel according to a number of parameters, such as how much system time it has already consumed and its status (perhaps waiting for I/O -- such processes are favored by the kernel). Linux gives you the ability to bias the kernel's priority algorithm, favoring certain processes over others.

The priority of a process can be determined by examining the PRI column in the results produced from issuing either the **top** or **ps -l** commands. The values displayed are relative; the higher the priority number, the more CPU time the kernel offers to the process. The kernel does this by managing a queue of processes. Those with high priority are given more time, and those with low priority are given less time. On a heavily loaded system, a process with a very low priority may appear stalled.

### **3.6.1 nice**

One of the parameters used by the kernel to assign process priority is supplied by the user and is called a *nice number*. The **nice** command<sup>[20]</sup> is used to assign a priority number to the process. It is so named because it normally causes programs to execute with lower priority levels than with their default. Thus, the process is being "nice" to other processes on the system by yielding CPU time. With this scheme, more "niceness" implies a lower priority, and less niceness implies a higher priority.

<sup>[20]</sup> Some shells, not including **bash**, have a built-in **nice** command.

By default, user processes are created with a *nice number* of zero. With this setting, **nice** doesn't sway the kernel's prioritization scheme one way or another. Positive numbers lower the priority, and negative numbers raise the priority. Nice numbers range from -20 to +19. Any user can start a process with a positive nice number, but only the superuser (*root*) can lower a process' nice number to raise its priority. For example, if you have a long-running utility program but don't want to impact interactive performance, a positive nice number will lower the job's priority and improve interactive performance.

## ***nice***

---

### **Syntax**

```
nice [-n adjustment] [command ]  
nice [-adjustment] [command ]
```

### **Description**

The **nice** command is used to alter another command's nice number at start time. For normal users, *adjustment* is an integer from 1 to 19. If you're the superuser, the *adjustment* range is from -20 to 19. If an *adjustment* number is not specified, the process' **nice** number defaults to 10. The *command* consists of any command that you might enter on the command line, including all options, arguments, redirections, and the background character **&**.

If both *adjustment* and *command* are omitted, nice displays the current scheduling priority, which is inherited.

### **Example 1**

The following command starts a program in the background with reduced priority, using the default nice number of 10:

```
$ nice somecmd -opt1 -opt2 arg1 arg2 &
```

### **Example 2**

As superuser, you can start programs with elevated priority. These equivalent commands start the **vi** editor with a higher priority, which may be necessary for administrative purposes if the system is exceptionally slow:

```
# nice --10 vi /etc/hosts.deny
# nice -n -10 vi /etc/hosts.deny
```

Note the double dash ( **-- 10** ) in the first form. The first dash indicates that an option follows, while the second dash indicates a negative number.

Be careful when using **nice** on interactive programs such as editors, word processors, or browsers. Assigning a program a positive nice number will most likely result in sluggish performance. Remember, the higher the positive number, the lower the resulting priority level.<sup>[21]</sup> For that reason, you should try not to assign positive nice numbers to foreground jobs on your terminal. If the system gets busy, your terminal could hang awaiting CPU time, which has been sacrificed by **nice**.

<sup>[21]</sup> The actual scheduling priority used by the kernel is dynamic and takes into account many more factors than the nice value.

### 3.6.1.1 Changing nice numbers on running processes

The **nice** command only works to change the nice number for new processes at the time that they're started. To modify a running program, use the **renice** command.

#### ***renice***

---

#### **Syntax**

```
renice [+|-]nicenumber [option] targets
```

#### **Description**

Alter the *nicenumber* to set the scheduling priority of one or more running *target* processes. By default, **renice** assumes that the *targets* are numeric PIDs. One or more *options* may also be used to interpret *targets* as processes owned by specific users.

#### **Frequently used options**

**-u**

Interpret *targets* as usernames, affecting all processes owned by those users.

**-p**

Interpret *targets* as PIDs (the default).

## Examples

This command will lower the priority of the process with PID 501 by increasing its **nice** number to the maximum:

```
$ renice 20 501
```

The following command can be used to increase the priority of all of user *jdean*'s processes as well as the process with PID 501:

```
# renice -10 -u jdean -p 501
```

In this command, **-10** indicates a **nice** value of negative 10, thus giving PID 501 a higher priority on the system. A dash isn't used for the **nice** value because the dash could be confused for an option, such as **-u**.

### ***On the Exam***

Be sure to know the range and meaning of **nice** numbers and how to change them for new and existing processes. Also note that **nice** and **renice** specify their numbers differently. With **nice**, a leading dash can indicate a nice number (e.g., **-10**), including a negative one with a second dash (e.g., **-- 10**). On the other hand, **renice** does not need the hyphen.

The **renice** command is handy for managing groups of processes, particularly to affect user processes by username. In addition, if you're observing processes in **top**, you may change nice numbers for them interactively from within the **top** program (discussed in Objective 5) by using the single-keystroke **r** command. You will be prompted for the PID of the process whose nice number you wish to change and for the new nice number (if you are the superuser, you may enter negative values). The new nice number will be displayed by **top** in the column labeled *NI* for the process you specify.

## 3.7 Objective 7: Making Use of Regular Expressions

In *Objective 3*, filename globbing with wildcards is described, which enables us to list or find files with common elements (i.e., filenames or file extensions) at once. File globs make use of special characters such as **\***, which have special meanings in the

context of the command line. There are a handful of shell wildcard characters understood by **bash**, enough to handle the relatively simple problem of *globbing* filenames. Other problems aren't so simple, and extending the glob concept into any generic text form (files, text streams, program string variables, etc.) can open up a wide new range of capability. This is done using *regular expressions*.

Two tools that are important for the LPIC Level 1 exams and that make use of regular expressions are **grep** and **sed**. These tools are useful for text searches. There are many other tools that make use of regular expressions, including the *awk*, Perl, and Python languages and other utilities, but you don't need to be concerned with them for the purpose of the LPIC Level 1 exams.

### 3.7.1 Using grep

A long time ago, as the idea of regular expressions was catching on, the line editor **ed** contained a command to display lines of a file being edited that matched a given regular expression. The command is:

```
g/regular expression/p
```

That is, "on a global basis, print the current line when a match for *regular expression* is found," or more simply, "global regular expression print." This function was so useful that it was made into a standalone utility named, appropriately, **grep**. Later, the regular expression grammar of **grep** was expanded in a new command called **egrep** (for "extended *grep*"). You'll find both commands on your Linux system today, and they differ slightly in the way they handle regular expressions. For the purposes of Exam 101, we'll stick with **grep**, which can also make use of the "extended" regular expressions when used with the **-E** option. You will find some form of **grep** on just about every Unix or Unix-like system available.

## **grep**

---

### Syntax

```
grep [options] regex [files]
```

### Description

Search *files* or standard input for lines containing a match to regular expression *regex*. By default, matching lines will be displayed and nonmatching lines will not be displayed. When multiple files are specified, **grep** displays the filename as a prefix to the output lines (use the **-h** option to suppress filename prefixes).

## Frequently used options

**-c**

Display only a count of matched lines, but not the lines themselves.

**-h**

Display matched lines, but do not include filenames for multiple file input.

**-i**

Ignore uppercase and lowercase distinctions, allowing *abc* to match both *abc* and *ABC*.

**-n**

Display matched lines prefixed with their line numbers. When used with multiple files, *both* the filename and line number are prefixed.

**-v**

Print all lines that *do not* match *regex*. This is an important and useful option. You'll want to use regular expressions, not only to *select* information but also to *eliminate* information. Using **-v** inverts the output this way.

## Examples

Since regular expressions can contain both metacharacters and literals, **grep** can be used with an entirely literal *regex*. For example, to find all lines in *file1* that contain either *Linux* or *linux*, you could use **grep** like this:

```
$ grep -i linux file1
```

In this example, the *regex* is simply "linux." The uppercase *L* in "Linux" is matched by the command-line option **-i**. This is fine for literal expressions that are common.

However, in situations in which *regex* includes regular expression metacharacters that are also shell special characters (such as **\$** or **\***), the *regex* must be quoted to prevent shell expansion and pass the metacharacters on to **grep**.

As a simplistic example of this, suppose you have files in your local directory named *abc*, *abc1*, and *abc2*. When combined with **bash**'s **echo** expression, the *abc\** wildcard expression lists all files that begin with *abc*, as follows:

```
$ echo abc*
```

```
abc abc1 abc2
```

Now suppose that these files contain lines with the strings *abc*, *abcc*, *abccc*, and so on, and you wish to use **grep** to find them. You can use the shell wildcard expression *abc\** to expand to all the *abc* files as displayed with **echo** above, and you'd use an identical regular expression *abc\** to find all occurrences of lines containing *abc*, *abcc*, *abccc*, etc. Without using quotes to prevent shell expansion, the command would be:

```
$ grep abc* abc*
```

After shell expansion, this yields:

```
$ grep abc abc1 abc2 abc abc1 abc2 # no!
```

This is *not* what you intended! **grep** would search for the literal expression *abc*, because it appears as the first command argument. Instead, quote the regular expression with single or double quotes to protect it:<sup>[22]</sup>

<sup>[22]</sup> The difference between single quotes and double quotes on the command line is subtle and is explained later in this section.

```
$ grep 'abc*' abc*
```

or:

```
$ grep "abc*" abc*
```

After expansion, both examples yield the same results:

```
$ grep abc* abc abc1 abc2
```

Now this is what you're after. The three files *abc*, *abc1*, and *abc2* will be searched for the regular expression *abc\**. It is good to stay in the habit of quoting regular expressions on the command line to avoid these problems -- they won't be at all obvious because the shell expansion is invisible to you unless you use the **echo** command.

### ***On the Exam***

The use of **grep** and its options is common. You should be familiar with what each option does, as well as the concept of piping the results of other commands into **grep** for matching.

### **3.7.2 Using sed**

In Objective 2, we introduce **sed**, the *stream editor*. In that section, we talk about how **sed** uses *addresses* to locate text upon which it will operate. Among the addressing mechanisms mentioned is the use of regular expressions delimited between slash characters. Let's recap how **sed** can be invoked.

## sed

---

### Syntax

```
sed [options] 'command1' [files]
sed [options] -e 'command1' [-e 'command2'] [files]
sed [options] -f script [files]
```

### Description

Note that *command1* is contained within single quotes. This is necessary for the same reasons as with **grep**. The text in *command1* must be protected from evaluation and expansion by the shell.

The address part of a **sed** command may contain regular expressions, which are enclosed in slashes. For example, to show the contents of *file1* except for blank lines, the **sed** delete (**d**) command could be invoked like this:

```
$ sed '/^$/ d' file1
```

In this case, the regular expression `^$` matches blank lines and the **d** command removes those matching lines from **sed**'s output.

#### 3.7.2.1 Quoting

As shown in the examples for **grep** and **sed**, it is necessary to *quote* regular expression metacharacters if you wish to preserve their special meaning. Failing to do this can lead to unexpected results when the shell interprets the metacharacters as file globbing characters. There are three forms of quoting you may use to preserve special characters:

##### \ (an unquoted backslash character)

By applying a backslash before a special character, it will not be interpreted by the shell but will be passed through unaltered to the command you're entering.

For example, the `*` metacharacter may be used in a regular expression like this:

```
$ grep abc\* abc abc1 abc2
```

Here, files *abc*, *abc1*, and *abc2* are searched for the regular expression *abc\**.

##### Single quotes

Surrounding metacharacters with the single-quote character also protects them from interpretation by the shell. All characters inside a pair of single quotes are assumed to have their literal value.

#### *Double quotes*

Surrounding metacharacters with the double-quote character has the same effect as single quotes, with the exception of the \$, ' (single quote), and \ (backslash) characters. Both \$ and ' retain their special meaning within double quotes. The backslash retains its special meaning when followed by \$, ', another backslash, or a newline.

In general, single quotes are safest for preserving regular expressions.

### ***On the Exam***

Pay special attention to quoting methods used to preserve special characters, because the various forms don't necessarily yield the same result.

### **3.7.3 Regular Expressions**

Linux offers many tools for system administrators to use for processing text. Many, such as **sed** and the *awk* and Perl languages, are capable of automatically editing multiple files, providing you with a wide range of text-processing capability. To harness that capability, you need to be able to define and delineate specific text segments from within files, text streams, and string variables. Once the text you're after is identified, you can use one of these tools or languages to do useful things to it.

These tools and others understand a loosely defined pattern language. The language and the patterns themselves are collectively called regular expressions (often abbreviated just *regexp* or *regex*). While regular expressions are similar in concept to file globs, many more special characters exist for regular expressions, extending the utility and capability of tools that understand them.

Regular expressions are the topic of entire books (such as Jeffrey E. F. Friedl's excellent and very readable *Mastering Regular Expressions*, published by O'Reilly & Associates). Exam 101 requires the use of simple regular expressions and related tools, specifically to perform searches from text sources. This section covers only the basics of regular expressions, but it goes without saying that their power warrants a full understanding. Digging deeper into the regular expression world is highly recommended when you have the chance.

#### **3.7.3.1 Regular expression syntax**

It would not be unreasonable to assume that some specification defines how regular expressions are constructed. Unfortunately, there isn't one. Regular expressions have been incorporated as a feature in a number of tools over the years, with varying degrees of consistency and completeness. The result is a cart-before-the-horse scenario, in which utilities and languages have defined their own flavor of regular expression syntax, each with its own extensions and idiosyncrasies. Formally defining the regular expression syntax came later, as did efforts to make it more consistent. Regular expressions are defined by arranging strings of text, or *patterns*. Those patterns are composed of two types of characters:

### *Metacharacters*

Like the special file *globbing* characters, regular expression *metacharacters* take on a special meaning in the context of the tool in which they're used. There are a few metacharacters that are generally thought of to be among the "extended set" of metacharacters, specifically those introduced into **egrep** after **grep** was created. Now, most of those can also be handled by **grep** using the **-E** option. Examples of metacharacters include the **^** symbol, which means "the beginning of a line," and the **\$** symbol, which means "the end of a line." A complete listing of metacharacters follows in [Table 3-6](#), [Table 3-7](#), and [Table 3-8](#).

### *Literals*

Everything that is not a metacharacter is just plain text, or literal text. It is often helpful to consider regular expressions as their own language, where literal text acts as words and phrases. The "grammar" of the language is defined by the use of metacharacters. The two are combined according to specific rules (which, as mentioned earlier, may differ slightly among various tools) to communicate ideas and get real work done. When you construct regular expressions, you use metacharacters and literals to specify three basic ideas about your input text:

### *Position anchors*

A position anchor is used to specify the position of one or more character sets in relation to the entire line of text (such as the beginning of a line).

### *Character sets*

A character set matches text. It could be a series of literals, metacharacters that match individual or multiple characters, or combinations of these.

### *Quantity modifiers*

Quantity modifiers follow a character set and indicate the number of times the set should be repeated. These characters "give elasticity" to a regular expression by allowing the matches to have variable length.

The next section lists commonly used metacharacters. The examples given with the metacharacters are very basic, intended just to demonstrate the use of the metacharacter in question. More involved regular expressions are covered later.

### 3.7.4 Regular Expression Examples

Now that the gory details are out of the way, here are some examples of simple regular expression usage that you may find useful.

#### 3.7.4.1 Anchors

Anchors are used to describe position information. [Table 3-6](#) lists anchor characters.

Table 3-6. Regular Expression Position Anchors	
Regular Expression	Description
<code>^</code>	Match at the beginning of a line. This interpretation makes sense only when the <code>^</code> character is at the lefthand side of the <i>regex</i> .
<code>\$</code>	Match at the end of a line. This interpretation makes sense only when the <code>\$</code> character is at the righthand side of the <i>regex</i> .

---

#### Example 1

Display all lines from *file1* where the string "Linux" appears at the start of the line:

```
$ grep '^Linux' file1
```

#### Example 2

Display lines in *file1* where the last character is an "x":

```
$ grep 'x$' file1
```

Display the number of empty lines in *file1* by finding lines with nothing between the beginning and the end:

```
$ grep -c '^$' file1
```

Display all lines from *file1* containing only the word "null" by itself:

```
$ grep '^null$' file1
```

#### 3.7.4.2 Groups and ranges

Characters can be placed into groups and ranges to make regular expressions more efficient, as shown in [Table 3-7](#).

<b>Regular Expression</b>	<b>Description</b>
[ abc ] [ a-z ]	Single-character groups and ranges. In the first form, match any single character from among the enclosed characters a, b, or c. In the second form, match any single character from among the range of characters bounded by a and z. The brackets are for grouping only and are not matched themselves.
[^abc] [^a-z]	Inverse match. Match any single character not among the enclosed characters a, b, and c or in the range a-z. Be careful not to confuse this inversion with the anchor character ^, described earlier.
\<word>	Match words. Words are essentially defined as being character sets surrounded by whitespace and adjacent to the start of line, the end of line, or punctuation marks. The backslashes are required and enable this interpretation of < and >.
. (the single dot)	Match any single character except a <i>newline</i> .
\	As mentioned in the section on quoting earlier, turn off (escape) the special meaning of the character that follows, turning metacharacters in to literals.

---

### Example 1

Display all lines from *file1* containing either "Linux," "linux," "TurboLinux," and so on:

```
$ grep '[Ll]inux' file1
```

### Example 2

Display all lines from *file1* which contain three adjacent digits:

```
$ grep '[0-9][0-9][0-9]' file1
```

### Example 3

Display all lines from *file1* beginning with any single character other than a digit:

```
$ grep '^[^0-9]' file1
```

### Example 4

Display all lines from *file1* that contain the whole word "Linux" or "linux," but not "LinuxOS" or "TurboLinux":

```
$ grep '\<[Ll]inux\>' file1
```

### Example 5

Display all lines from *file1* with five or more characters on a line (excluding the newline character):

```
$ grep '.....' file1
```

### Example 6

Display all nonblank lines from *file1* (i.e., that have at least one character):

```
$ grep '.' file1
```

### Example 7

Display all lines from *file1* that contain a period (normally a metacharacter) using escape:

```
$ grep '\.' file1
```

#### 3.7.4.3 Modifiers

Modifiers change the meaning of other characters in a regular expression. [Table 3-8](#) lists these modifiers.

<b>Regular Expression</b>	<b>Description</b>
*	Match an unknown number (zero or more) of the single character (or single-character <i>regex</i> ) that precedes it.
?	Match zero or one instance of the preceding <i>regex</i> . This modifier is an "extended" feature and available in <b>grep</b> only when the <b>-E</b> command-line option is used.
+	Match one or more instances of the preceding <i>regex</i> . This modifier is an "extended" feature and available in <b>grep</b> only when the <b>-E</b> command-line option is used.
\{ <i>n,m</i> \}	Match a range of occurrences of the single character or <i>regex</i> that precedes this construct. \{ <i>n</i> \} matches <i>n</i> occurrences, \{ <i>n</i> ,\} matches at least <i>n</i> occurrences, and \{ <i>n,m</i> \} matches any number of occurrences between <i>n</i> and <i>m</i> , inclusively. The backslashes are required and enable this interpretation of { and }.
	Alternation. Match either the <i>regex</i> specified before <i>or</i> after the vertical bar. This modifier is an "extended" feature and available in <b>grep</b> only when the <b>-E</b> command-line option is used.

---

### Example 1

Display all lines from *file1* that contain "ab," "abc," "abcc," "abccc," and so on:

```
$ grep 'abc*' file1
```

### Example 2

Display all lines from *file1* that contain "abc," "abcc," "abccc," and so on, but not "ab":

```
$ grep 'abcc*' file1
```

### Example 3

Display all lines from *file1* that contain two or more adjacent digits:

```
$ grep '[0-9][0-9][0-9]*' file1
```

### Example 4

Display lines from *file1* that contain "file" (because ? can match zero occurrences), *file1*, or *file2*:

```
$ grep -E 'file[12]?' file1
```

### Example 5

Display all lines from *file1* containing at least one digit:

```
$ grep -E '[0-9]+' file1
```

### Example 6

Display all lines from *file1* that contain "111," "1111," or "11111" on a line by itself:

```
$ grep '^1\{3,5\}$' file1
```

### Example 7

Display all lines from *file1* that contain any three-, four-, or five-digit number:

```
$ grep '\<[0-9]\{3,5\}\>' file1
```

### Example 8

Display all lines from *file1* that contain "Happy," "happy," "Sad," "sad," "Angry," or "angry":

```
$ grep -E '[Hh]appy|[Ss]ad|[Aa]ngry' file1
```

#### 3.7.4.4 Basic regular expression patterns

---

##### Example 1

Match any letter:

```
[A-Za-z]
```

##### Example 2

Match any symbol (not a letter or digit):

```
[^0-9A-Za-z]
```

##### Example 3

Match an uppercase letter, followed by zero or more lowercase letters:

```
[A-Z][a-z]*
```

##### Example 4

Match a U.S. Social Security Number (123-45-6789) by specifying groups of three, two, and four digits separated by dashes:

```
[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}
```

##### Example 5

Match a dollar amount, using an escaped dollar sign, zero or more spaces or digits, an escaped period, and two more digits:

```
\\$[ 0-9]*\\. [0-9]\{2\}
```

##### Example 6

Match the month of June and its abbreviation, "Jun." The question mark matches zero or one instance of the *e* :

June?

### 3.7.4.5 Using regular expressions as addresses in sed

These examples are commands you would issue to **sed**. For example, the commands could take the place of *command1* in this usage:

```
$ sed [options] 'command1' [files]
```

These commands could also appear in a standalone **sed** script.

---

#### Example 1

Delete blank lines:

```
/^$/d
```

#### Example 2

Delete any line that doesn't contain #keepme::

```
/#keepme/!d
```

#### Example 3

Delete lines containing only whitespace (spaces or tabs). In this example, *tab* means the single tab character and is preceded by a single space:

```
/^[ tab]*$/d
```

#### Example 4

Delete lines beginning with periods or pound signs:

```
/^[. #]/d
```

#### Example 5

Substitute a single space for any number of spaces wherever they occur on the line:

s/ \*/ /g

### Example 6

Substitute *def* for *abc* from line 11 to 20, wherever it occurs on the line:

11,20s/abc/@@@/g

### Example 7

Translate the characters *a*, *b*, and *c* to the @ character from line 11 to 20, wherever they occur on the line:

11,20y/abc/@@@/

### ***On the Exam***

Make certain you are clear about the difference between *file globbing* and the use of regular expressions.